

Python for Data Science

4. Data Structures



Contents

4.1 **Lists and Tuples**

4.2 **Dictionaries and Sets**

4.3 **Strings**

4.4 **Files**

1

Lists

Lists

Lists typically store **homogeneous data**, that is, values of the *same* data type.

They also may store **heterogeneous data**, that is, data of many different types.

```
In [1]: c = [-45, 6, 0, 72, 1543]
```

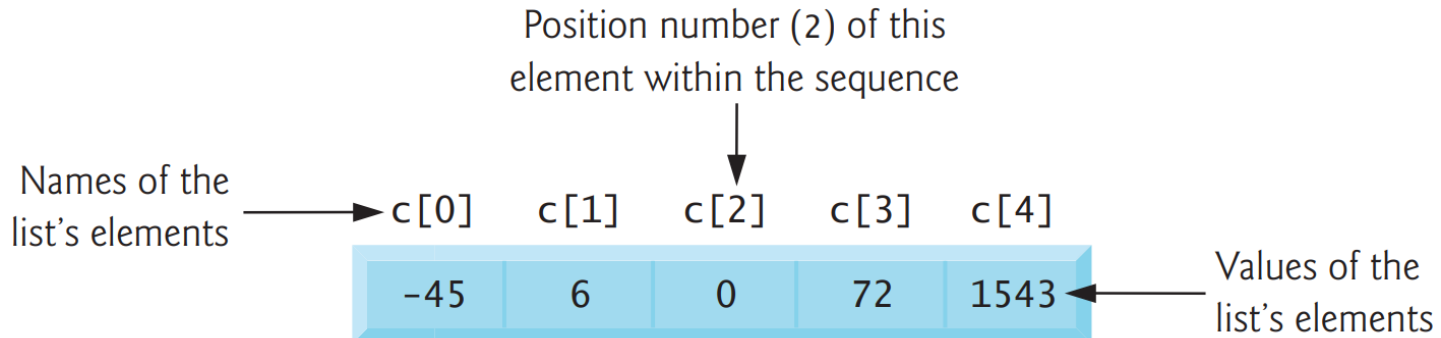
```
In [2]: c
```

```
Out[2]: [-45, 6, 0, 72, 1543]
```

```
['Mary', 'Smith', 3.57, 2022]
```

Accessing Elements of a List

You reference a list element by writing the list's name followed by the element's **index** (its **position number**) enclosed in square brackets ([]), the **subscription operator**).



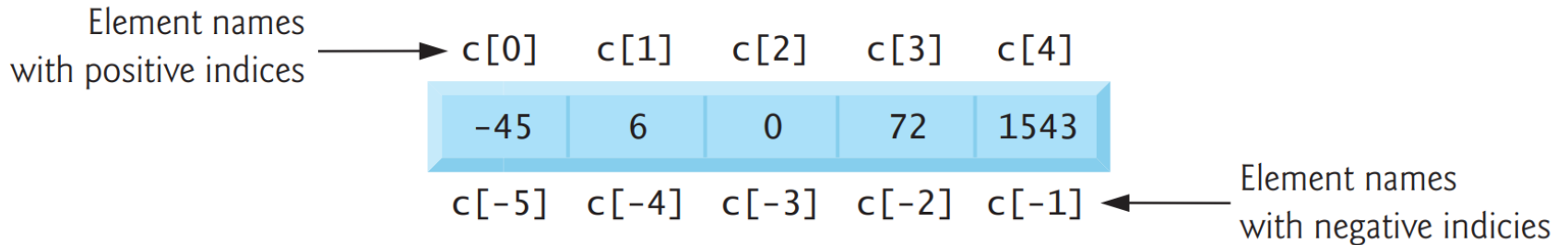
1 Lists

List's Length

To get a list's length, use the built-in `len` function.

Accessing Elements

Lists also can be accessed from the end by using *negative indices*.



Lists Are Mutable

Their elements can be modified.

```
In [11]: c[4] = 17
```

```
In [12]: c
```

```
Out[12]: [-45, 6, 0, 72, 17]
```

1

Lists

Some Sequences Are Immutable

Python's string and tuple sequences are *immutable*. You can get the individual characters in a string, but attempting to assign a new value to one of the characters causes a `TypeError`.

```
In [13]: s = 'hello'
```

```
In [14]: s[0]
```

```
Out[14]: 'h'
```

```
In [15]: s[0] = 'H'
```

`TypeError`

Traceback (most recent call last)

<ipython-input-15-812ef2514689> in <module>()

```
----> 1 s[0] = 'H'
```

`TypeError: 'str' object does not support item assignment`

1

Lists

Access a Nonexistent Element

Using an out-of-range list, tuple or string index causes an `IndexError`.

```
In [16]: c[100]
```

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-19-9a31ea1e1a13> in <module>()  
----> 1 c[100]
```

```
IndexError: list index out of range
```

List Elements in Expressions

List elements may be used as variables in expressions.

```
In [17]: c[0] + c[1] + c[2]
```

```
Out[17]: -39
```

1

Lists

Appending to a List

When the left operand of `+=` is a list, the right operand must be an *iterable*; otherwise, a `TypeError` occurs.

If the right operand contains multiple elements, `+=` appends them all.

```
In [18]: a_list = []
```

```
In [19]: for number in range(1, 6):  
...:     a_list += [number]  
...:
```

```
In [20]: a_list  
Out[20]: [1, 2, 3, 4, 5]
```

```
In [21]: letters = []
```

```
In [22]: letters += 'Python'
```

```
In [23]: letters  
Out[23]: ['P', 'y', 't', 'h', 'o', 'n']
```

Concatenating Lists

You can **concatenate** two lists, two tuples or two strings using the `+` operator. The result is a *new* sequence of the same type containing the left operand's elements followed by the right operand's elements. The original sequences are unchanged.

A `TypeError` occurs if the `+` operator's operands are different sequence types.

```
In [24]: list1 = [10, 20, 30]
```

```
In [25]: list2 = [40, 50]
```

```
In [26]: concatenated_list = list1 + list2
```

```
In [27]: concatenated_list
```

```
Out[27]: [10, 20, 30, 40, 50]
```


1

Lists

Accessing List

List elements also can be accessed via their indices and the subscription operator (`[]`).

The function call `range(len(concatenated_list))` produces a sequence of integers representing `concatenated_list`'s indices (in this case, 0 through 4). When looping in this manner, you must ensure that indices remain in range.

```
In [28]: for i in range(len(concatenated_list)):
...:     print(f'{i}: {concatenated_list[i]}')
...:
0: 10
1: 20
2: 30
3: 40
4: 50
```

1

Lists

Comparison Operators

You can compare entire lists element-by-element using comparison operators.

```
In [29]: a = [1, 2, 3]
```

```
In [30]: b = [1, 2, 3]
```

```
In [31]: c = [1, 2, 3, 4]
```

```
In [32]: a == b # True: corresponding elements in both are equal  
Out[32]: True
```

```
In [33]: a == c # False: a and c have different elements and lengths  
Out[33]: False
```

```
In [34]: a < c # True: a has fewer elements than c  
Out[34]: True
```

```
In [35]: c >= b # True: elements 0-2 are equal but c has more elements  
Out[35]: True
```

1 Tuples

Tuples

Tuples are immutable and typically store heterogeneous data, but the data can be homogeneous.

Creating Tuples

To create an empty tuple, use empty parentheses.
You can pack a tuple by separating its values with commas.

```
In [1]: student_tuple = ()
```

```
In [2]: student_tuple
```

```
Out[2]: ()
```

```
In [3]: len(student_tuple)
```

```
Out[3]: 0
```

```
In [4]: student_tuple = 'John', 'Green', 3.3
```

```
In [5]: student_tuple
```

```
Out[5]: ('John', 'Green', 3.3)
```

```
In [6]: len(student_tuple)
```

```
Out[6]: 3
```

1 Tuples

Accessing Tuple Elements

Like list indices, tuple indices start at 0.

Assigning a value to a tuple element causes a `TypeError`.

```
In [11]: time_tuple = (9, 16, 1)
```

```
In [12]: time_tuple
```

```
In [13]: time_tuple[0] * 3600 + time_tuple[1] * 60 + time_tuple[2]  
Out[13]: 33361
```

Adding Items

As with lists, the `+=` augmented assignment statement can be used with strings and tuples, even though they're immutable. Concatenating a tuple to a tuple1 creates a new tuple.

```
In [14]: tuple1 = (10, 20, 30)
```

```
In [15]: tuple2 = tuple1
```

```
In [16]: tuple2  
Out[16]: (10, 20, 30)
```

```
In [17]: tuple1 += (40, 50)
```

```
In [18]: tuple1  
Out[18]: (10, 20, 30, 40, 50)
```

```
In [19]: tuple2  
Out[19]: (10, 20, 30)
```

1

Tuples

Appending Tuples to Lists

```
In [20]: numbers = [1, 2, 3, 4, 5]
```

```
In [21]: numbers += (6, 7)
```

```
In [22]: numbers
```

```
Out[22]: [1, 2, 3, 4, 5, 6, 7]
```

Tuples may contain mutable objects. Even though the tuple is immutable, its list element is mutable.

```
In [23]: student_tuple = ('Amanda', 'Blue', [98, 75, 87])
```

```
In [24]: student_tuple[2][1] = 85
```

```
In [25]: student_tuple
```

```
Out[25]: ('Amanda', 'Blue', [98, 85, 87])
```

Unpacking Sequences

```
In [1]: student_tuple = ('Amanda', [98, 85, 87])
```

```
In [2]: first_name, grades = student_tuple
```

```
In [3]: first_name
```

```
Out[3]: 'Amanda'
```

```
In [4]: grades
```

```
Out[4]: [98, 85, 87]
```

1

Tuples

Swapping Values

You can swap two variables' values using sequence packing and unpacking.

```
In [11]: number1 = 99
```

```
In [12]: number2 = 22
```

```
In [13]: number1, number2 = (number2, number1)
```

```
In [14]: print(f'number1 = {number1}; number2 = {number2}')  
number1 = 22; number2 = 99
```

Accessing Safely with `enumerate`

The preferred mechanism for accessing an element's index and value is the built-in function `enumerate`.

This function receives an iterable and creates an iterator that, for each element, returns a tuple containing the element's index and value.

```
In [15]: colors = ['red', 'orange', 'yellow']
```

```
In [16]: list(enumerate(colors))
```

```
Out[16]: [(0, 'red'), (1, 'orange'), (2, 'yellow')]
```

1

Lists and Tuples

Sequence Slicing

You can **slice** sequences to create new sequences of the same type containing *subsets* of the original elements. Slice operations can modify mutable sequences.

```
In [1]: numbers = [2, 3, 5, 7, 11, 13, 17, 19]
```

```
In [2]: numbers[2:6]
```

```
In [7]: numbers[:]
```

```
Out[7]: [2, 3, 5, 7, 11, 13, 17, 19]
```

```
Out[3]: [2, 3, 5, 7, 11, 13]
```

```
In [4]: numbers[0:6]
```

```
Out[4]: [2, 3, 5, 7, 11, 13]
```

```
In [5]: numbers[6:]
```

```
Out[5]: [17, 19]
```

```
In [6]: numbers[6:len(numbers)]
```

```
Out[6]: [17, 19]
```

```
In [7]: numbers[:]
```

```
Out[7]: [2, 3, 5, 7, 11, 13, 17, 19]
```

1

Lists and Tuples

Slicing with Steps

```
In [8]: numbers[::2]
Out[8]: [2, 5, 11, 17]
```

Slicing with Negative Indices and Steps

```
In [9]: numbers[::-1]
Out[9]: [19, 17, 13, 11, 7, 5, 3, 2]
```

```
In [10]: numbers[-1:-9:-1]
Out[10]: [19, 17, 13, 11, 7, 5, 3, 2]
```

Modifying Lists Via Slices

```
In [11]: numbers[0:3] = ['two', 'three', 'five']
```

```
In [12]: numbers
Out[12]: ['two', 'three', 'five', 7, 11, 13, 17, 19]
```

```
In [13]: numbers[0:3] = []
```

```
In [14]: numbers
Out[14]: [7, 11, 13, 17, 19]
```


2

Dictionaries

Creating a Dictionary

You can create a dictionary by enclosing in curly braces, {}, a comma-separated list of key–value pairs, each of the form *key: value*.

You can create an empty dictionary with {}.

Because dictionaries are *unordered* collections, the display order can differ from the order in which the key–value pairs were added to the dictionary.

```
In [1]: country_codes = {'Finland': 'fi', 'South Africa': 'za',  
...:                    'Nepal': 'np'}  
...:
```

```
In [2]: country_codes  
Out[2]: {'Finland': 'fi', 'South Africa': 'za', 'Nepal': 'np'}
```

2

Dictionaries

Empty Dictionary

The built-in function `len` returns the number of key-value pairs in a dictionary.

You can use a dictionary as a condition to determine if it's empty—a non-empty dictionary evaluates to `True`.

An empty dictionary evaluates to `False`. To demonstrate this, in the following code we call method `clear` to delete the dictionary's key-value pairs.

```
In [5]: country_codes.clear()
```

```
In [6]: if country_codes:
...:     print('country_codes is not empty')
...: else:
...:     print('country_codes is empty')
...:
country_codes is empty
```

2

Dictionaries

Iterating through a Dictionary

Multiple keys can have the same value.

Dictionary method **items** returns each key-value pair as a tuple.

```
In [1]: days_per_month = {'January': 31, 'February': 28, 'March': 31}
```

```
In [2]: days_per_month
```

```
Out[2]: {'January': 31, 'February': 28, 'March': 31}
```

```
In [3]: for month, days in days_per_month.items():  
...:     print(f'{month} has {days} days')  
...:
```

```
January has 31 days
```

```
February has 28 days
```

```
March has 31 days
```

2

Dictionaries

```
In [1]: roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 100}
```

```
In [2]: roman_numerals
```

```
Out[2]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 100}
```

Accessing the Value Associated with a Key

```
In [3]: roman_numerals['V']
```

```
Out[3]: 5
```

```
In [4]: roman_numerals['X'] = 10
```

```
In [5]: roman_numerals
```

```
Out[5]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 10}
```

Adding a New Key-Value Pair

```
In [6]: roman_numerals['L'] = 50
```

```
In [7]: roman_numerals
```

```
Out[7]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 10, 'L': 50}
```

2

Dictionaries

Removing a Key-Value Pair

```
In [8]: del roman_numerals['III']
```

```
In [9]: roman_numerals
```

```
Out[9]: {'I': 1, 'II': 2, 'V': 5, 'X': 10, 'L': 50}
```

```
In [10]: roman_numerals.pop('X')
```

```
Out[10]: 10
```

```
In [11]: roman_numerals
```

```
Out[11]: {'I': 1, 'II': 2, 'V': 5, 'L': 50}
```

Attempting to Access a Nonexistent Key

```
In [13]: roman_numerals.get('III')
```

```
In [14]: roman_numerals.get('III', 'III not in dictionary')
```

```
Out[14]: 'III not in dictionary'
```

```
In [15]: roman_numerals.get('V')
```

```
Out[15]: 5
```

2

Dictionaries

Dictionary Methods keys and values

```
In [1]: months = {'January': 1, 'February': 2, 'March': 3}
```

```
In [2]: for month_name in months.keys():  
...:     print(month_name, end=' ')  
...:  
January February March
```

```
In [3]: for month_number in months.values():  
...:     print(month_number, end=' ')  
...:  
1 2 3
```

Converting Dictionary to Lists

```
In [9]: list(months.keys())  
Out[9]: ['January', 'February', 'March', 'December']
```

```
In [10]: list(months.values())  
Out[10]: [1, 2, 3, 12]
```

```
In [11]: list(months.items())  
Out[11]: [('January', 1), ('February', 2), ('March', 3)]
```

Sets

A set is an unordered collection of *unique* values.

Sets may contain only immutable objects, like strings, `ints`, `floats` and tuples that contain only immutable elements.

Though sets are iterable, they are not sequences and do not support indexing and slicing with square brackets, `[]`.

Creating a Set

An important use of sets is **duplicate elimination**, which is automatic when creating a set.

Also, the resulting set's values are *not* displayed in the same order as they were listed.

```
In [1]: colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'}
```

```
In [2]: colors
```

```
Out[2]: {'blue', 'green', 'orange', 'red', 'yellow'}
```

2

Sets

Determining a Set's Length

```
In [3]: len(colors)
Out[3]: 5
```

Checking Whether a Value Is in a Set

```
In [4]: 'red' in colors
Out[4]: True
```

```
In [5]: 'purple' in colors
Out[5]: False
```

```
In [6]: 'purple' not in colors
Out[6]: True
```

Iterating Through a Set

```
In [7]: for color in colors:
...:     print(color.upper(), end=' ')
...:
RED GREEN YELLOW BLUE ORANGE
```

set Function

```
In [8]: numbers = list(range(10)) + list(range(5))
```

```
In [9]: numbers
Out[9]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4]
```

```
In [10]: set(numbers)
Out[10]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```


Union

```
In [1]: {1, 3, 5} | {2, 3, 4}
Out[1]: {1, 2, 3, 4, 5}
```

```
In [2]: {1, 3, 5}.union([20, 20, 3, 40, 40])
Out[2]: {1, 3, 5, 20, 40}
```

Intersection

```
In [3]: {1, 3, 5} & {2, 3, 4}
Out[3]: {3}
```

```
In [4]: {1, 3, 5}.intersection([1, 2, 2, 3, 3, 4, 4])
Out[4]: {1, 3}
```

Difference

```
In [5]: {1, 3, 5} - {2, 3, 4}
Out[5]: {1, 5}
```

```
In [6]: {1, 3, 5, 7}.difference([2, 2, 3, 3, 4, 4])
Out[6]: {1, 5, 7}
```

Symetric Difference

```
In [7]: {1, 3, 5} ^ {2, 3, 4}
Out[7]: {1, 2, 4, 5}
```

```
In [8]: {1, 3, 5, 7}.symmetric_difference([2, 2, 3, 3, 4, 4])
Out[8]: {1, 2, 4, 5, 7}
```

Union Augmented Assignment

```
In [1]: numbers = {1, 3, 5}
```

```
In [2]: numbers |= {2, 3, 4}
```

```
In [3]: numbers
```

```
Out[3]: {1, 2, 3, 4, 5}
```

update

```
In [4]: numbers.update(range(10))
```

```
In [5]: numbers
```

```
Out[5]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

add

```
In [6]: numbers.add(17)
```

```
In [7]: numbers.add(3)
```

```
In [8]: numbers
```

```
Out[8]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 17}
```

remove

```
In [9]: numbers.remove(3)
```

```
In [10]: numbers
```

```
Out[10]: {0, 1, 2, 4, 5, 6, 7, 8, 9, 17}
```

Introduction

Python views a **text file** as a sequence of characters and a **binary file** (for images, videos and more) as a sequence of bytes.

As in lists and arrays, the first character in a text file and byte in a binary file is located at position 0, so in a file of n characters or bytes, the highest position number is $n - 1$.

For each file you **open**, Python creates a **file object** that you'll use to interact with the file.



End of File

Every operating system provides a mechanism to denote the end of a file.

Some represent it with an **end-of-file marker** (as in the preceding figure), while others might maintain a count of the total characters or bytes in the file.

Programming languages generally hide these operating-system details from you.

Standard File Objects

- `sys.stdin`—the standard input file object
- `sys.stdout`—the standard output file object, and
- `sys.stderr`—the standard error file object.

4

Files

Writing to a Text File

```
In [1]: with open('accounts.txt', mode='w') as accounts:
...:     accounts.write('100 Jones 24.98\n')
...:     accounts.write('200 Doe 345.67\n')
...:     accounts.write('300 White 0.00\n')
...:     accounts.write('400 Stone -42.16\n')
...:     accounts.write('500 Rich 224.62\n')
...:
```

Reading from a Text File

```
In [1]: with open('accounts.txt', mode='r') as accounts:
...:     print(f'{"Account":<10}{"Name":<10}{"Balance":>10}')
...:     for record in accounts:
...:         account, name, balance = record.split()
...:         print(f'{account:<10}{name:<10}{balance:>10}')
...:
```

Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62